by
Jeff Prosise

Tutor

UNDERSTANDING HIMEM.SYS

What's the purpose of the HIMEM.SYS driver that comes with *Windows* 3.0, and how do you use it? Does it somehow enable you to move TSRs and device drivers into high memory under DOS?

E. I. Muehldorf
Potomac, Maryland

■ UNDERSTANDING HIMEM.SYS: Here's how Microsoft's XMS driver enables access to memory above 640K.

PC MAGAZINE Remember the discussion of extended memory and the Extended Memory Specification (XMS) that appeared in the December 26, 1989, Tutor column? HIMEM.SYS is Microsoft's version of the XMS driver. *Windows* 3.0 uses it to access extended memory in a well-behaved manner that's compatible with other extended memory-aware programs, yet not necessarily with EMS-handlers.

Because HIMEM.SYS is present, you could conceivably run other XMS processes concurrently with *Windows* in memory, without putting either *Windows* or the other programs at risk.

HIMEM.SYS marshals access to extended memory (or, to be more specific, *all* memory above 640K) the same way EMS drivers control expanded memory: it provides a set of functions for programs to access via far calls to the driver's entry point. You obtain the entry point by executing an interrupt 2Fh with AX set to 4310h; the 32-bit address of the driver's entry point is returned in ES:BX. These control functions, which are summarized in Figure 1, enable programs to allocate blocks of memory, move data in and out of them, and release them when they're no longer needed. To understand the functions, you need to know that HIMEM.SYS recognizes three types of memory blocks:

■ Upper Memory Blocks, or UMBs, which lie at addresses between 640K and 1MB.

■ The High Memory Area, or HMA, the first 64K of memory (minus 16 bytes) beyond 1MB.

■ Extended Memory Blocks, or EMBs, which may lie anywhere in extended memory above the HMA.

The Upper Memory area, between 640K and 1MB, is reserved for use by the system's video display and BIOS, but chunks of it normally remain unused. HIMEM.SYS relies on other utilities, known as UMB providers, to create Upper Memory Blocks for backfilling these unused regions of memory. When these blocks are present, they can be accessed in real mode, since they fall within the real-mode addressing limits of the 80*x*86 family.

Functions 10h and 11h arbitrate accesses to UMBs so that two processes contending for the same block won't destroy each other by overwriting what the other placed there. Function 10h allocates a block; function 11h releases it. While a block is allocated to a program, it is protected from other applications that abide by XMS protocols, because the driver will not allocate the same block to two programs.

HIMEM.SYS carves Extended Memory Blocks out of the region of extended memory above the 1,088K mark (1,024K plus the 64K occupied by the HMA). Functions 08h through 0Fh govern access

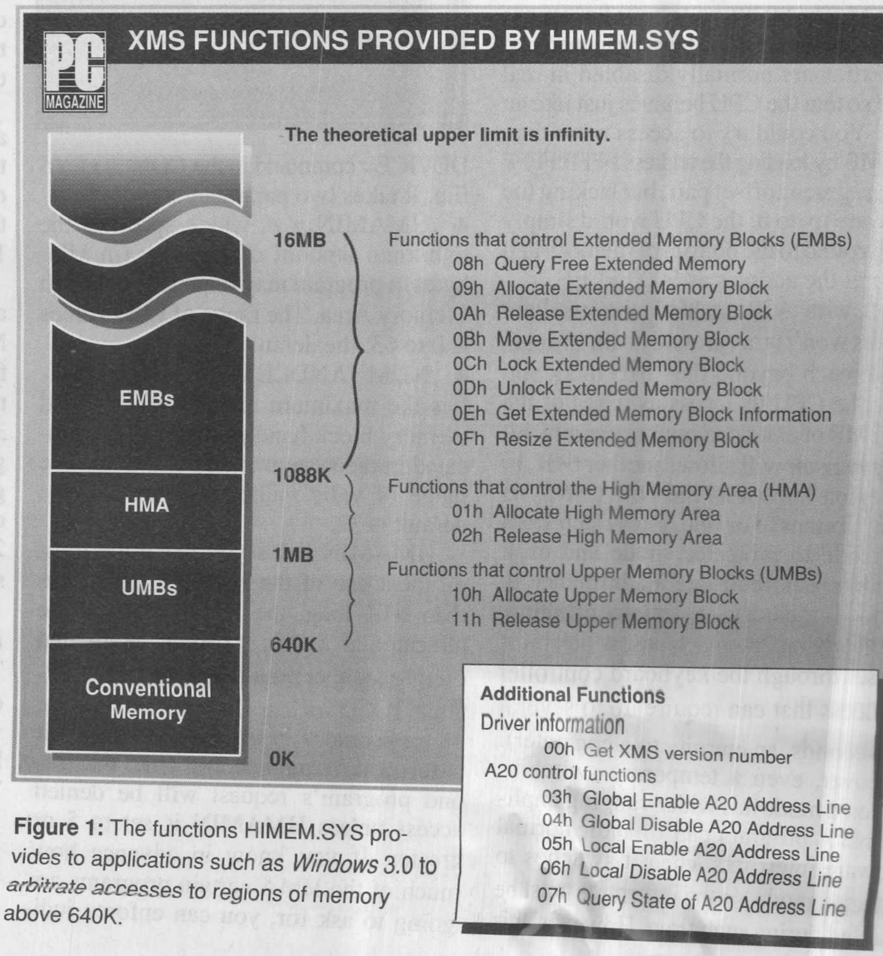

Figure 1: The functions HIMEM.SYS provides to applications such as *Windows* 3.0 to arbitrate accesses to regions of memory above 640K.

to these blocks. For example, function 0Bh, Move Extended Memory Block, moves data from conventional memory to an EMB, from an EMB to conventional memory, or between two EMBs. A program may also access extended memory directly by calling function 0Ch to obtain a 32-bit linear address for the EMB, simultaneously locking its location in memory. Function 0Dh is the counterpart to 0Ch, used to unlock a previously locked block.

HIMEM.SYS also provides hardware-independent control over the A20 address line, which allows processes running on 286s, 386s, and 486s to access the first 64K of extended memory (the HMA) in real mode.

The High Memory Area is set apart from the rest of extended memory because of a design quirk that allows 286s and 386s to access it in real mode. The secret? Selectively enabling the A20 address line so that a 21-bit memory address can be formed. Recall that 8088s and 8086s have 20 address lines (numbered A0 through A19) that allow them to access up to 1MB of RAM. The 80286 and above have more address lines for reaching higher into memory. The extras are normally disabled in real mode so that the CPU behaves just like an 8088. You could try to access more than just 1MB by loading the address FFFF:FFFF into a segment:offset pair, but lacking the hardware to do it, the CPU would simply wrap around to the bottom of memory and interpret the address as 0000:FFEF.

But with A20 enabled, the resultant address won't wrap around at all. Instead, it will reach beyond the 1MB mark and allow the CPU to access as much of the next 1MB of extended memory as a 16-bit offset register will allow: another 64K, or the region known as the HMA. Why is this important? For one, the switch from real mode to protected mode and back normally required to gain admittance to extended memory is time-consuming, particularly on 286 machines, which must be reset through the keyboard controller (a process that can require up to several milliseconds, an eternity to a computer). Moreover, even a temporary switch to protected mode has wider-ranging implications. A program can't handle normal hardware interrupts when it switches to protected mode, so the interrupts must be disabled during such time. However, this means that your program may miss certain interrupts if it flips into protected mode and back, so avoid this.

The A20 line provides a bit of relief from the confines of real mode's 1MB cap on memory, and the XMS driver's arbitration of the A20 line ensures that two XMS-aware programs contending for it won't interfere with each other.

HIMEM.SYS provides five functions, numbered 03h through 07h, for dealing with the A20 line. *Global* enable/disable is to be used by programs that have requested and been allocated the HMA. *Local* enable/disable is for programs that do not own the HMA. It is each program's responsibility to enable the A20 line before it accesses the HMA and to restore the original state of the line (which is obtainable with function 07h) after it's done.

You install HIMEM.SYS with a DEVICE= command in the CONFIG.SYS file. It takes two parameters:

The A20 line provides a bit of relief from the confines of real mode's 1MB cap on memory.

- /HMAMIN = *n,* which specifies the minimum amount of memory (in kilobytes) a program may request in the High Memory Area. The range of valid values is 0 to 63; the default is 0.
- /NUMHANDLES = *n,* which specifies the maximum number of extended memory block handles that may be allocated in the system at any one time. The range of valid values is 0 to 128; the default is 32.

HMAMIN helps you make the most efficient use of the HMA. Slightly less than 64K long, the HMA can only be allocated as a unit; it cannot be divvied up into smaller chunks as the rest of extended memory can. If a program needing 4K of memory in the HMA requests it before a program needing 40K, the second program's request will be denied access unless HMAMIN is set to 5 or greater. If you know in advance how much of the HMA certain programs are going to ask for, you can enforce judicious use of it by tweaking HMAMIN accordingly.

NUMHANDLES, the second parameter, specifies how many EMB handles HIMEM.SYS should reserve room for internally. Each additional handle requires 6 bytes of conventional memory. As a rough rule of thumb, set aside space for 16 handles per megabyte of extended memory in your system. Then all of extended memory can be put to use in chunks as small as 64K.

Both HMAMIN and NUMHANDLES are defined in Version 2.0 of the XMS specification, dated August 23, 1988. The version of HIMEM.SYS shipped with *Windows* 3.0 accepts two additional switches on the DEVICE= line: /SHADOW:ON|OFF and /MACHINE:NAME.

Many PCs provide a feature known as *ROM shadowing,* where ROM, which is notoriously slow, is copied to RAM and its code is executed from there for increased speed. By default, HIMEM.SYS attempts to disable ROM shadowing on these PCs to free additional extended memory. The /SHADOW switch lets you explicitly instruct HIMEM.SYS to enable or disable shadowing. This switch is useless on some systems: Many hardware configurations do not permit shadowing to be turned on and off under software control.

The /MACHINE switch is a means of adapting HIMEM.SYS to the peculiarities of certain systems. At present, the only name it accepts is `acer1100`, which tells HIMEM.SYS it's running on an Acer 1100 PC.

Can HIMEM.SYS be used to load TSRs and device drivers into high memory? No—at least not by itself. The XMS specification does not provide control functions like those used by programs such as *386MAX* and *QEMM386* to stuff programs into high memory. The best way to get programs out of conventional memory is still a 386 memory manager or, on 286s with expanded memory, a utility similar to Quarterdeck Systems' *QRAM*.

ASK THE TUTOR

The Tutor solves practical problems and explains techniques for using your hardware and software more productively. To have your questions answered, write to Tutor, *PC Magazine,* One Park Avenue, New York, NY 10016, or upload them to PC MagNet (see page 8 for access instructions). We're sorry, but we're unable to answer questions individually. ■

instruction and then proceeds to read the control word back. If the NDP is an 8087, the IEM bit will be set; if it's a 287 or 387, it will not.

The final test differentiates between 287s and 387s. It relies on FINIT initializing the 287 to use *projective* infinity, where positive and negative infinity are equal, but initializes the 387 to use *affine* infinity—where positive and negative infinity appear at opposite ends of the number line.

In fact, the 387 uses affine infinity only; it ignores the setting of the control word's Infinity Control (IC) bit, which switches between affine and projective infinity on the 8087 and 287.

In order to gauge which infinity control mode is in effect after FINIT is executed, WHATNDP pushes a real 1 followed by a real 0 onto the NDP's internal register stack and divides 1 by 0. The result, stored in register ST(0), is positive infinity. It then copies the positive infinity to register ST(1) and generates a negative infinity in ST(0) by changing the sign with the FCHS instruction. Finally, it compares ST(0) and ST(1) with an FCOMPP instruction. If they're equal, then the infinity mode is projective (positive infinity and negative infinity are the same) and the coprocessor must therefore be a 287. However, if the two values are not equal, then the infinity mode is affine and the coprocessor must be a 387.

One thing to watch for is that if you run WHATNDP on a 486, it will tell you there's a 387 coprocessor installed. Of course there's really not. The 486 has an on-chip NDP built right into it that's architecturally similar to a 387. Obviously, if you run WHATCPU and find that there's a 486 installed, there's no need to even run WHATNDP.

ASK THE TUTOR

The Tutor solves practical problems and explains techniques for using your hardware and software more productively. Questions about DOS and systems in general are answered here. To have your questions answered, write to Tutor, *PC Magazine,* One Park Avenue, New York, NY 10016, or upload them to PC MagNet (see the ''By Modem'' sidebar in the Utilities column). We're sorry, but we're unable to answer questions individually. ■